

高级语言程序设计

第 5 章 函数与变量

华中师范大学物理学院 李安邦

- 本章主要介绍 C/C++ 语言中与函数和变量相关的知识，讨论一些程序整体结构有关的问题。
- 对正确理解 C/C++ 语言/正确书写 C/C++程序都很重要。
- 是学习用 C/C++ 程序设计时应了解的“深层问题”。
- 函数的定义与使用，函数原型
- 变量类，作用域与存在期
- 预处理命令，命名空间，多文件项目开发

第5章 函数与变量

5.1 函数的定义与调用

5.1.1 对自定义函数的需求

5.1.2 函数定义

5.1.3 函数的调用

5.1.4 函数和程序

5.1.5 局部变量的作用域和生存期

5.1.6 函数调用的参数传递机制

5.2 程序的函数分解

5.3 循环与递归

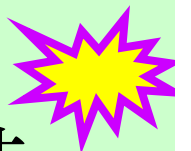
5.4 外部变量与静态局部变量

5.5 声明与定义

5.6 预处理

5.7 程序动态除错方法（二）

- 函数可看作是C/C++语言基本功能的扩充。
- 函数是特定计算过程的抽象，具有一定通用性，可以按规定方式对具体数据使用。对一个（或一组）具体数据，函数执行可以计算出一个结果，这个结果可以在后续计算中使用。
- 函数的作用是使人可以把一段计算抽象出来，封装（包装）起来，使之成为程序中的一个独立实体。还有为这样封装起的代码取一个名字，做成一个函数定义(function definition)。
- 当程序中需要做这段计算时，可以通过一种简洁的形式要求执行这段计算，这种片段称为函数调用(function calling)。



函数抽象机制的意义：

- 重复片段可用唯一的函数定义和一些形式简单的函数调用取代，使程序更简短清晰。
- 同样计算片段只描述一次，易于修改。
- 函数定义和使用形成对复杂程序的分解。可独立考虑函数定义与使用，大大提高工作效率。
- 具有独立逻辑意义的函数可看作高层基本操作，使人可以站在合适的抽象层次上观察把握程序的意义。

认识函数调用

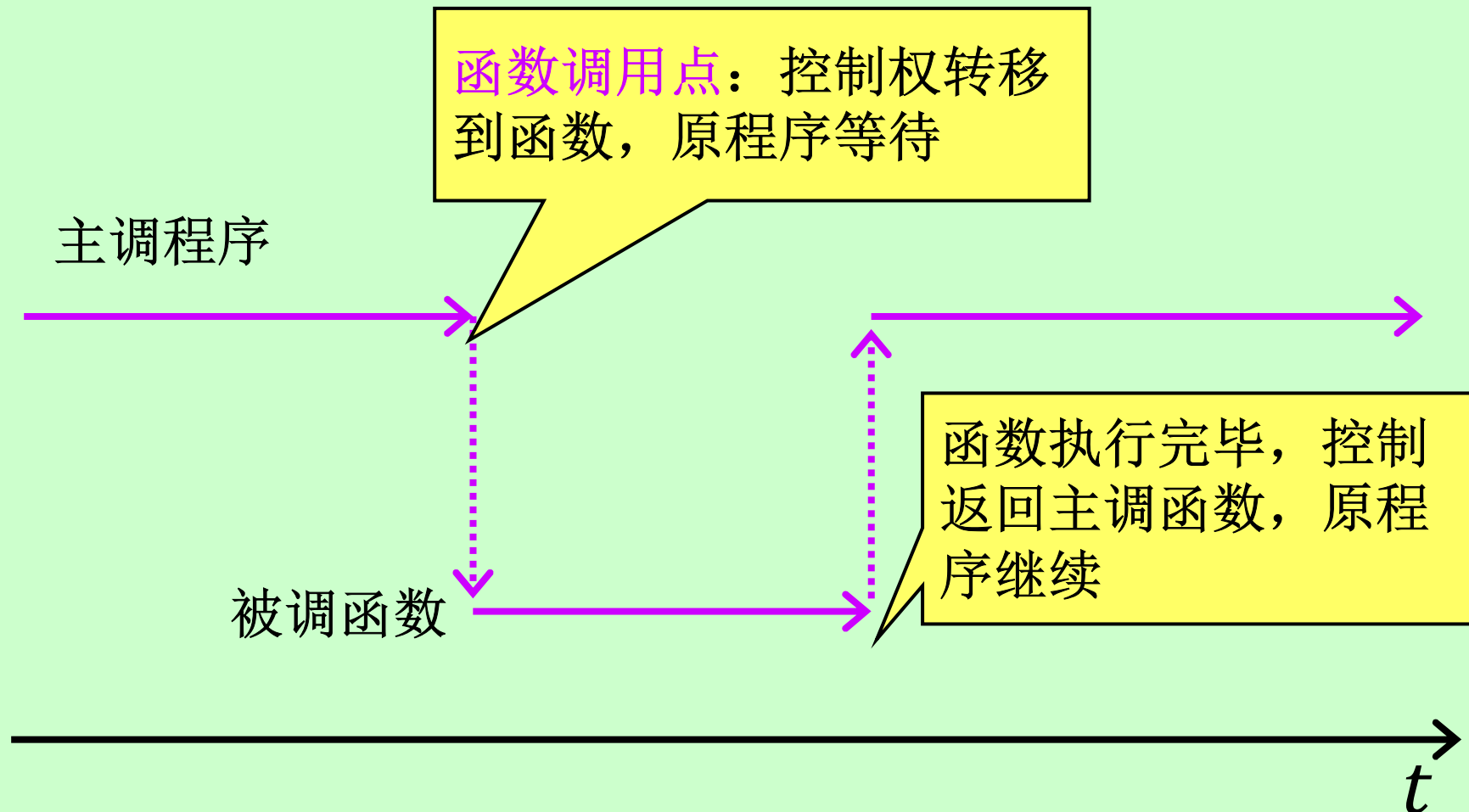


图5-1 函数的调用、执行与返回

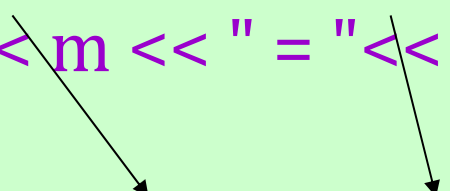
5.1.1 对自定义函数的需求

虽然系统提供了大量的标准库函数供用户使用，但是在实际程序设计中，标准库函数还不能满足用户的需求，存在着许多对特定函数的需求，这里举两个简单的例子。

【例5-1】哥德巴赫猜想是数论中的一个著名的难题，它的陈述为“任一大于2的偶数都可写成两个质数之和”。这个难题的严格证明需要高深的数学理论，至今还没有得到彻底解决。请写程序在小范围内来验证这一猜想：对6到200之间的各个偶数找出一种质数分解，即找出两个质数，满足两者之和等于这个偶数。

程序主要部分在主体结构上大致上可以写成这样：

```
int main() {  
    int m, n;  
    for (m = 6; m <= 2000; m += 2)  
        for (n = 3; n <= m/2; n += 2) {  
            if ( n 是质数 && m-n是质数) {  
                cout << m << " = " << n << " + " << m-n << endl;  
                break;  
            }  
        }  
    return 0;  
}
```



能否把以前写过的程序
片段拿过来使用？

【例5-2】在第4章中有一个简单猜数游戏（见“4.4.1 编程实例1：一个简单猜数游戏”），整个程序的工作流程是简洁明了的，但是写出的程序超过了 80 行，在阅读源代码的时候，读者可能难以把握整个程序的工作流程。

如果把整个程序拆分成几个不同的部分，就能使主程序变得简洁明了。

同时我们还注意到，在输入最大值和输入用户猜测数据时，都分别花多条语句来处理输入出错的情形，而这些语句在功能实际上是重复的。

以上例子说明了编程人员有自己定义函数的需要。也分别说明了需要对自定义函数需求的两种场合：

- 1、需要多次重复使用某个计算片段，
- 2、把较长的程序进行合理拆分，从而使主程序变得简单易读，方便把握整个程序的工作流程。

5.1.2 函数定义

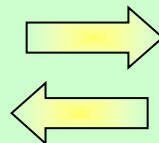
要使用自己定义的函数，**必须把函数定义的代码段包含在整个程序里**，这样的一段代码称为一个“**函数定义**”。

在程序里有了某个函数的定义后，就可以在程序里任何需要它的地方写出**调用**语句来使用它们。

用户可以在程序中自己**定义(define)**函数，然后就可以在程序中对函数进行**调用(call)**。

对函数的定义和调用**是互相照应的**：在调用时需要按照定义时所规定的语法形式书写调用语句，在定义里需要按照调用时所需的功能进行设计。

定义(define)



调用(call)

函数定义的形式

函数定义的形式：函数头部 函数体



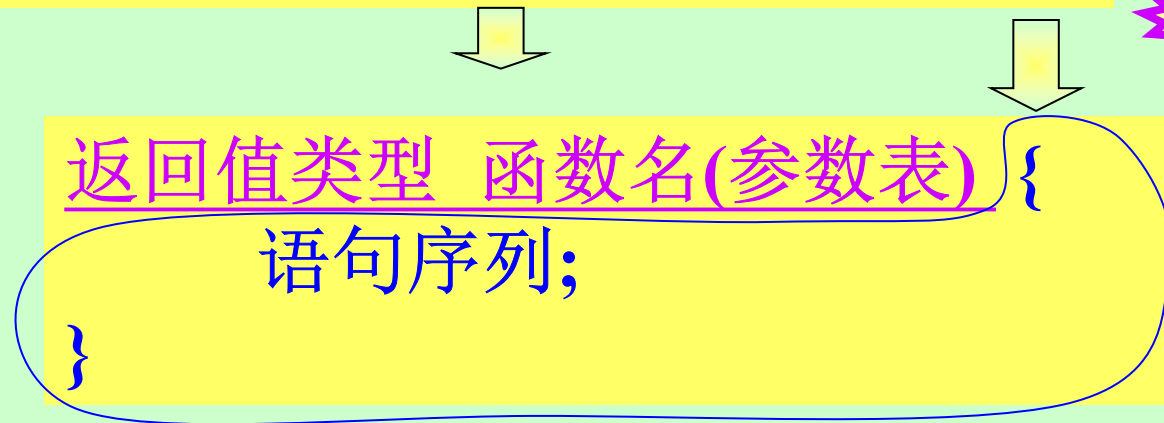
返回值类型 函数名(参数表)

描述函数执行结束时
将会返回的值的类型，
也可以是 **void**。

用标识符表示，
供以后调用这个
函数时使用；

声明参数的个数、各参数的类型和参数名。
参数名是为了在函数里使用实际参数的值。

函数定义的形式： 函数头部 函数体



函数体 (body)：用 {} 包括起来的复合结构。

其中定义的变量是本函数的局部变量。

函数头部中的参数也视为局部变量来使用。

函数体里的特殊语句：return（返回）语句。
该语句使函数结束。

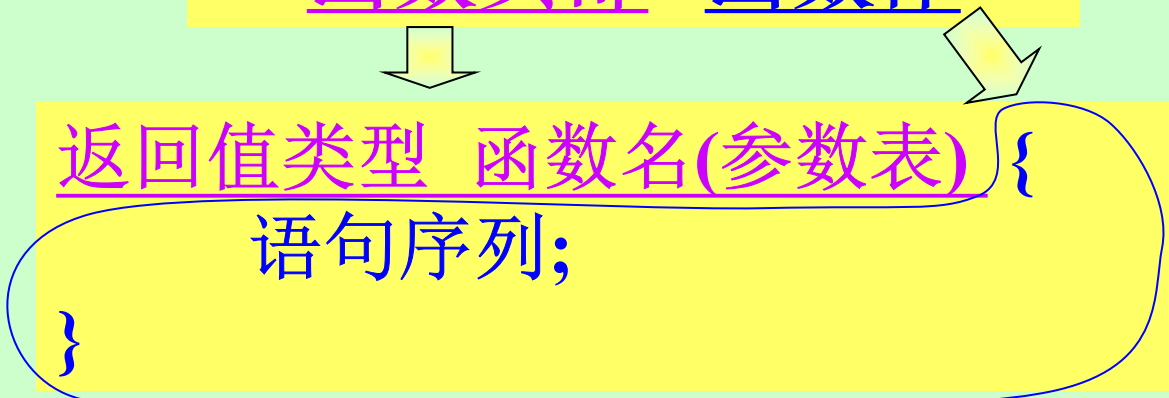
用法1:  返回值类型 函数名(参数表) {
 语句序列;
 return 表达式;
}

先算表达式，以其值作为函数返回值。

用法2:  void 函数名(参数表) {
 语句序列;
 return;
}

直接从函数中返回。

函数头部 函数体



```
返回值类型 函数名(参数表) {  
    语句序列;  
}
```

定义函数时，需要先分析程序中的需求进行设计：

- 准备拿几个什么样的参数来进行计算？
- 计算完成之后要返回什么样的值？
- 然后给函数起一个合适的名字。

按这样设计来写好函数头部之后，就可以在函数体中编写相应的语句来完成所需的功能。

【例5-3】编写一个函数，用于在给定半径时计算圆面积。

准备拿几个什么样的参数来进行计算？ double radius

计算完成之后要返回什么类型的值？ double

然后给函数起一个合适的名字。 scircle

```
double scircle (double radius) { //版本1
```

```
    return 3.14159265 * radius * radius;
```

```
}
```

函数头部中的参数也视为局部变量来使用

```
double scircle (double radius) { //版本2
```

```
    double erea = 3.14159265 * radius * radius;
```

```
    return erea;
```

```
}
```

函数体中定义的变量是本函数的局部变量。

【例5-4】 编写一个函数，用于在给定矩形的长度和宽度时计算矩形面积。

```
double srect(double a, double b) { //两个参数  
    return a * b;  
}
```

【例5-5】 编写一个函数，在屏幕上输出 20 个星号并换行。

```
void prtStar() { //无参数，无返回值  
    cout << "*****" << endl;  
    return; //返回（无返回值）  
}
```

函数头部 函数体

```
double scircle (double radius) { ... }
```

```
double srect(double a, double b) { ... }
```

```
void prtStar() { ... }
```

上面三个简单的例子，分别说明了单个参数/多个参数/无参数、有返回值/无返回值的函数的写法。

当然，参数和返回值的情况可以随意组合，例如写出有参数但无返回值的函数、或无参数但是有返回值的函数。

5.1.3 函数的调用

已经定义好的函数就可以在程序中进行调用了。

在表达式中使用函数的形式是：先写函数名，然后写一对圆括号（无参函数也需要写），再根据函数定义时的函数头部中所规定的参数类型和参数个数写上单个/多个表达式（用逗号隔开）。这些表达式是送给函数作为计算对象的，称为函数的实际参数，简称实参。

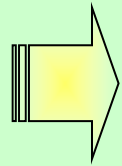
所以，函数调用的一般形式为：

函数名(实际参数)

函数名(实际参数, 实际参数)

函数名()

.....



函数调用：函数名(实际参数表)



多个参数用逗号分隔。



在函数定义时，参数表中的参数称为形参。

在调用时，把实参的值传递给形参。

函数体的复合语句在参数具有特定实参值的情况下开始执行。

在编写调用语句时，应该根据函数定义时的函数头部中的参数表和返回值进行相应的书写：

1、参数表非空，则调用时必须提供个数正确、类型合适的实参。实参是具体函数计算的出发点。实参可以是数值、变量或由数值和变量构成的表达式。

如果函数的参数表为空，那么就不需要（而且也不允许）提供参数，只需要写一对空的圆括号（不可省略）。

2、如果提供的实参类型与形参类型不一致，那么在执行时就会发生类型转换。

3、对于具有返回值的函数，在其中执行到某一条 `return` 表达式; 语句时，该语句中的表达式的值被计算出来并作为该函数的返回值，这个返回值可供调用点处后续使用（也可以闲弃不用），所以，有返回值的函数一般出现在表达式里，用其返回值参与后续操作（例如给其它变量赋值，或参与后续计算，或者直接打印输出）。

当然，无返回值的函数在执行结束时没有任何值可供调用处使用，显然不能放在表达式里使用，即不能用于做赋值、计算或打印之类的操作。

例如，对于 `scircle` 和 `srect` 函数（它们分别需要1个和2个参数，都有 `double` 类型的返回值），可以写出如下的调用语句：

```
double s;  
s = scircle(2.4); //直接提供数值作为参数，返回值用于赋值;  
s = scircle(2.4 + sin(1.57)); //含有数学函数的表达式作为参数，返回值  
    用于赋值;  
cout << scircle(1.5 + 2.4); //算术表达式作为参数，返回值用于打印输出;  
double r = 1.5;  
s = scircle(r); //变量作为参数，返回值用于赋值;  
cout << scircle(r * 2); //算术表达式作为参数，返回值用于打印输出  
double length = 3.5, width = 4.2;  
s = srect(3.5, 4.2); //直接提供数值作为参数，返回值用于赋值;  
s = srect(3 * sin(2.), 2 * cos(5.2)); //以表达式作为参数，返回值用于赋值;  
s = srect(length, width); //变量作为参数，返回值用于赋值;  
cout << srect(length, width); //变量作为参数，返回值用于打印输出
```

而对于 prtStar 函数，由于它不需要参数，所以在调用时就不需要提供参数，可以这样调用（注意，小括号不能省略）：

```
prtStar();
```

如果故意提供参数给它，那么语句就是错误的；

```
prtStar(100); //wrong!
```

而且这个函数没有返回值，所以就没有返回值可供用于赋值或打印。如下语句都是错误的：

```
s = prtStar(); //wrong!
```

```
cout << prtStar(); //wrong!
```


【例5-6】 把前文的几个示例函数写在同一个程序文件中，并写一个 main 函数，在其中调用这些函数。

```
#include <iostream>
#include <cmath>
using namespace std;
```

```
//double scircle (double radius) { //计算圆面积函数之版本1
//  return 3.14159265 * radius * radius;
//}
```

```
double scircle (double radius) { //计算圆面积函数之版本2
    double erea = 3.14159265 * radius * radius;
    return erea;
}
```

```
double srect (double a, double b) {  
    return a * b;  
}
```

```
void prtStar() {  
    cout << "*****" << endl;  
    return;  
}
```

```
int main() {  
    double s;  
  
    prtStar();  
    s = scircle(2.4);  
    cout << "s= " << s << endl;  
    s = scircle(2.4 + sin(1.57));  
    cout << "s= " << s << endl;  
    cout << scircle(1.5+2.4) << endl;
```

注意：

- 1、自定义的函数要写在 main 函数上方；
- 2、函数之间要写适当的空行，清晰美观。（函数内部也可以写适当的空行）。

```
double r = 1.5;
s = scircle(r);
cout << "s= " << s << endl;
cout << scircle(r * 2) << endl;
prtStar();

s = srect(3.5, 4.2);
cout << "s= " << s << endl;
s = srect(3 * sin(2.), 2 * cos(5.2));
cout << "s= " << s << endl;
double length = 3.5, width = 4.2;
s = srect(length, width);
cout << "s= " << s << endl;
cout << "s= " << srect (length, width) << endl;
prtStar();

return 0;
}
```

上面这个例题包含了很多知识，下面几节逐一展开介绍。

5.1.4 函数和程序

一个完整的程序，必须有且仅有一个名为 main 的函数（主函数）。

```
int main () {  
    .....  
    return 0;  
}
```

函数 main 表示程序的执行过程。程序从 main 的体开始执行，直到该复合结构结束。

其他函数不经调用就不会执行。main 在程序启动时被自动调用（由运行系统调用）。

程序里不允许调用 main。

- 在书写的形式上，一个程序文件中的每个函数都是平等的，彼此不能包含，不能把一个函数的定义写在另一个函数内部。
- 在习惯上，人们常把自定义的函数写在前面，把 `main` 函数写在最后。这是为了满足对函数的“先定义后使用”规则。
- 在一个程序中，不允许出现多个自定义函数具有相同的返回值类型、函数名和参数表的情况。

5.1.5 局部变量的作用域和生存期

一个变量定义，是定义了一个具有特定类型的变量，并给变量命名。

同时，一个变量定义还确定了两个问题：

- 1、在程序中的哪个范围内该变量定义有效。每个变量都有一个确定的作用范围，称为该变量的作用域(scope)，变量的作用域由变量定义的位置确定。
- 2、变量的实现基础是内存单元，变量在程序运行中建立，并在某个时间撤消。一个变量在程序执行中从建立到撤消的存在时期称为这个变量的生存期(lifetime)或存在期。

作用域和生存期是程序语言中的两个重要概念，弄清楚它们，许多问题就容易理解了。

作用域和生存期有联系但又不同，这两个概念是分别从空间和时间的角度来体现变量的特性。

作用域讲变量定义的作用范围，说的是源程序中的一段范围，可以在代码中划清楚，是静态概念。

生存期则完全是动态概念，讲的是程序执行过程中的一段期间。变量在生存期里一直保持着自己的存储单元，保存于这些存储单元中的值在被赋新值之前会一直保持不变。

在C/C++程序中的任何复合语句里的任何位置都可以定义变量。在一个复合结构里定义的变量可以在该复合结构的内部使用。



从作用域的角度来看，在这些语句中所定义的变量只能在相应的局部范围内使用。它们的作用域是从该变量定义的语句开始，到复合语句结束为止，在这个复合语句之外该定义无效。因此，这些变量被称为局部变量（Local variables）。

函数形参都看作函数定义的局部变量，其作用域就是这个函数的函数体。

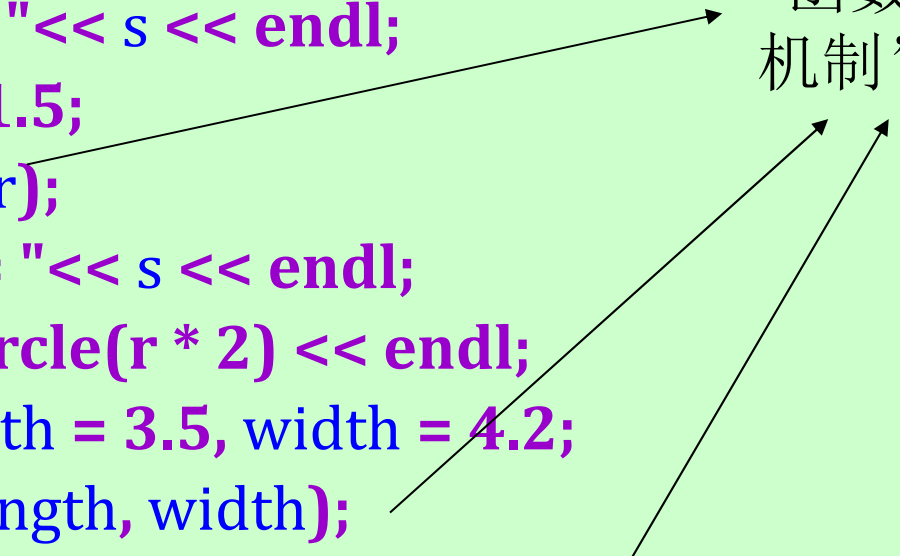
for 语句的小括号中定义的变量，作用域就是整个for 语句。

```
#include <iostream>
using namespace std;

//double scircle (double radius) {
//    return 3.14159265 * radius * radius;
//}
double scircle (double radius) {
    double erea = 3.14159265 * radius * radius;
    return erea;
}
double srect (double a, double b) {
    return a * b;
}
void prtStar() {
    cout << "*****" << endl;
    return;
}
```

```
int main() {  
    double s;  
    prtStar();  
    s = scircle(2.4);  
    cout << "s= " << s << endl;  
    double r = 1.5;  
    s = scircle(r);  
    cout << "s= " << s << endl;  
    cout << scircle(r * 2) << endl;  
    double length = 3.5, width = 4.2;  
    s = srect(length, width);  
    cout << "s= " << srect (length, width) << endl;  
    return 0;  
}
```

“函数调用的参数传递
机制” 见下一节。

Three arrows originate from the function calls in the code: one from 'scircle(2.4)', one from 'scircle(r)', and one from 'srect (length, width)'. All three arrows point towards the text '“函数调用的参数传递机制” 见下一节。’.

不同作用域内的变量名是否允许同名呢？语言对此有如下规定：

(1) 同一作用域里不允许定义两个以上同名变量，也就是说，作用域相同的变量的名字不能冲突。否则使用哪个变量的问题就无法确定了。

(2) 不同作用域容许定义同名变量。也是人们经常做的。

【例5-7】 写一个函数求整数平方和 $\sum_{n=1}^m n^2$ ，然后在 main 函数中调用这个函数求出给定m的值。

```
int sumsq(int m) {  
    int sum = 0;  
    for (int n=0; n < m; n++) {  
        int k= n*n;  
        sum = sum + k;  
        cout << "n= " << n << " sum= " << sum << endl;  
    }  
    cout << "m= " << m << " sum= " << sum << endl;  
    return sum;  
}  
  
int main() {  
    int m;  
    cout << "input an integer: ";  
    cin >> m;  
    cout << "sum= " << sumsq(m) << endl;  
    return 0;  
}
```

函数形参的作用域是
这个函数的函数体。

for 语句的小括号中
定义的变量的作用域
就是整个 for 语句。

局部变量的作用域，是从
该变量定义的语句开始，
到复合语句结束为止。

由于在函数体内可以嵌套其它复合语句，因此产生了作用域的嵌套，在这些嵌套的作用域中是否可以使用同名变量呢？

这时，这两个同名变量虽然同名但作用域不同，所以互不相干，故仍然服从上面第二条规定：不同作用域中容许定义同名变量。但此时有一个新问题：使用该变量名时到底是在使用哪一个变量？语言对此还有一条规定：

(3) 当内层复合语句出现同名变量定义时，外层同名定义将被内层定义遮蔽。也就是说，在使用该变量名时，实际上是优先使用内层定义的变量。

例如，本例中的sumsq函数中，把变量“k”的名字写成“m”也可以：

```
int sumsq(int m) { //正确而让人难懂的版本
    int sum = 0;
    for (int n = 0; n < m; n++) {
        int m = n*n; //定义了同名变量m
        sum = sum + m; //使用内层变量m
        cout << "n= " << n << " sum= " << sum << endl;
    }
    cout << "m= " << m << " sum=" << sum << endl; //使用形参m
    return sum;
}
```

这个函数是正确的，但是写成这样显然很让人难以读懂！因此，本书作者觉得对读者有用的建议是：**在程序中尽量避免在嵌套的作用域中出现同名变量！**

上机操作：

- 例题 5-6
- 习题5-1， 5-2 （写在同一个程序文件中： xxxx-xxx-prog5-1-2.cpp）
- 习题 4-13

所有源程序都要提交。

有时候也需要注意语句的写法有误而导致出现嵌套的变量遮蔽现象。例如下面的 sumsq 函数就含有功能性错误：

```
int sumsq(int m) { //含有错误的版本
    int sum; //定义了函数内的局部变量sum
    for (int n = 0, sum = 0; n < m; n++) { //定义了内层局部变量n和sum
        int k = n * n;
        sum = sum + k;
        cout << "n= " << n << " sum= " << sum << endl;
    }
    cout << "m= " << m << " sum=" << sum << endl;
    return sum;
}
```

这个函数的错误原因是：

上面详细介绍了变量的作用域，下面介绍变量的存在期。

在复合语句里定义的局部变量的存在期，就是这个复合语句的执行期间。



也就是说，该复合语句开始执行时建立这里面定义的所有变量。它们一直存在到该复合语句结束。复合语句结束时，内部定义的所有变量都撤消。

如果执行再进入这一复合语句，那么就再次建立这些变量。新建变量与上次执行建立的变量毫无关系，是另一组变量。

正是由于在复合语句里定义的变量被自动建立和撤消的性质，语言中也把它们称作自动变量。

这几句话很简单，但含义很深刻。

以 sumsq 函数为例说明：

```
int sumsq(int m) {  
    int sum = 0;           for结构开始执行时新建变量 n  
    for (int n=0; n < m; n++) {  
        int k= n*n;        //for结构每次执行循环体时新建变量k  
        sum = sum + k;  
        cout << "n= " << n << " sum= " << sum << endl;  
    } //for结构每次循环体执行结束时销毁变量k  
    //for结构执行结束时销毁新建变量 n  
    cout << "m= " << m << " sum=" << sum << endl;  
    return sum;  
}
```

在编程时应当注意变量的作用域和存在期，从而理解它们在特定执行环境中的值。如果不能正确理解这一点，则所写的程序可能会含有非常隐蔽的错误。假设有下面程序片段：

```
for (int n = 1; n < 10; n++) {  
    int k;  
    if (n == 1)  
        k = 5;  
    k = k + n; //循环执行第二次到达这里时k的值无法确定  
    cout << "k= " << k << endl;  
}
```

每次循环体开始执行时建立一个名为 k 的新变量（系统为它分配存储空间）。

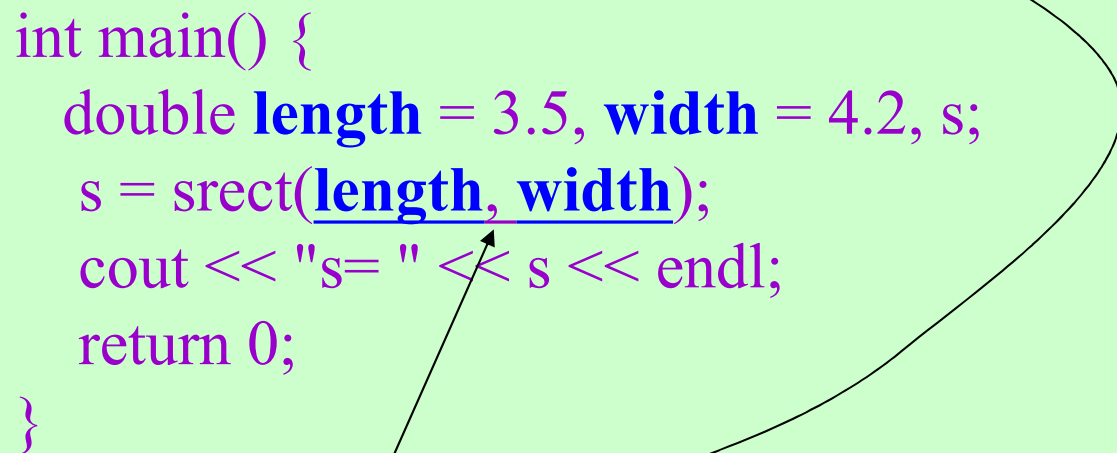
第一次循环时，由于 n 值为 1，k 赋值 5，执行 $k = k + n$ ；之后，k 的值为 6，循环结束时变量 k 被撤消。但在第二次及其后的循环执行中条件不成立，相应赋值语句不执行，这样到了 $k = k + n$ ；这一句时，新建立的变量 k 未经过赋值，值不能确定。所以这个程序中含有错误。

- 读者如果多次运行这个程序，通常会发现所输出的结果是一模一样的，最后输出的结果在数学上也是正确的（“ $k=50$ ”），好像这个程序并不含有错误似的。
- 这是因为系统在多次新建变量时，偶然地选用了上一次的内存空间，上一次的残留值就被用作初始值来使用。
- 如果系统工作繁忙，内存空间的存储情况频繁地发生变化，那么此程序运行再次进入循环并新建变量 k 时，很可能该变量所分配的内存空间与上一次并不相同，就会得到一个无法预料的价值作为初始值（用于执行 $k = k + n$; 语句）。所得的计算结果就很可能完全无法预料了。——因此，上面的程序片段中确实含有非常隐蔽的错误。

5.1.6 函数调用的参数传递机制

```
double srect (double a, double b) {  
    return a * b;  
}
```

```
int main() {  
    double length = 3.5, width = 4.2, s;  
    s = srect(length, width);  
    cout << "s= " << s << endl;  
    return 0;  
}
```

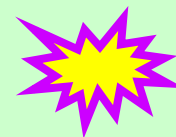


在调用函数时，实参与形参具体是什么样的关系？

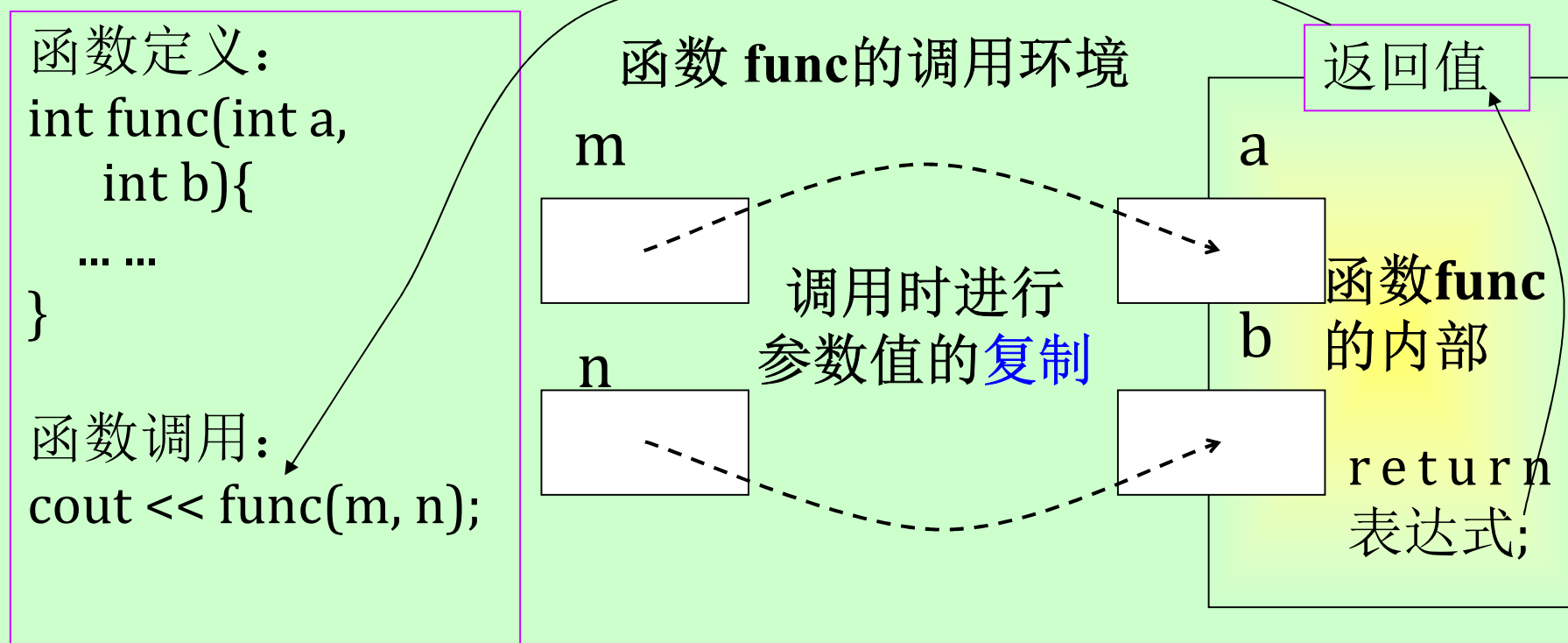
当实参是变量时，在函数体内改变之后的形参值是否会返回给实参呢？

这就需要我们理解C和C++语言中的参数机制。

C 和C++语言中的函数的基本参数机制是值参数：
函数调用时先计算实参表达式的值，把值复制给
对应形参，而后执行函数体。



函数内对形参的操作与实参无关。函数内对形参
的赋值与实参无关。



函数调用与参数值的传递

【例5-8】：

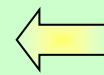
```
#include <iostream>
using namespace std;
void swap(int a, int b) {
    int k=a;  a=b;  b=k;
    cout << "swap: a=" << a << ", b=" << b << endl;;
}
int main() {
    int m=10, n=25;
    cout << "before: m=" << m << ", n=" << n << endl;
    swap(m, n);
    cout << "after: m=" << m << ", n=" << n << endl;
}
```

输出结果（**a** 和 **b** 的值在调用前后并未改变）：

before: m=10, n=25

swap: a=25, b=10

after: m=10, n=25



请仔细体会函数的
参数值传递机制！

C++中的引用

在C++语言中添加了一个名为“引用(reference)”的新特性，使用这种方式在函数之间传递参数，可以在函数中改变调用处的多个变量的值。

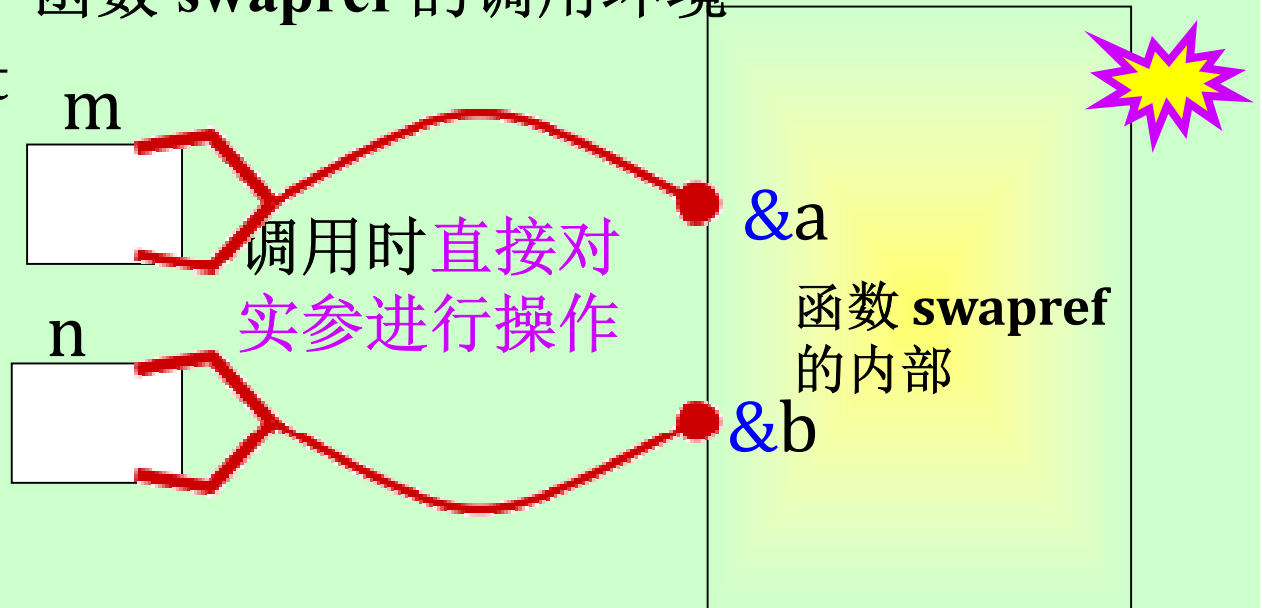
函数定义：

```
int swapref(int &a, int  
&b){  
    ... ..  
}
```

函数调用：

```
swapref(m, n);
```

函数 swapref 的调用环境



此处 & 表示引用
(不是取地址)

举例：

```
void swapref(int &a, int &b) { //形参a, b都是引用
    int k=a;  a=b;  b=k;
    cout << "swaped inside: a= " << a << " b= " << b << endl;
}
```

```
int main() {
    int m=10, n=25;
    cout << "before swapref: m= " << m << " n= " << n <<
    endl;
    swapref(m, n);
    cout << "after swapref: m= " << m << " n= " << n << endl;
}
```

输出结果（a 和 b 的值在调用前后确实改变了）：

before swapref: m=10, n=25

swaped inside: a=25, b=10

after swapref: m=25, n=10

```
void swap(int a, int b) { ... }
```

```
void swapref(int &a, int &b) { ... }
```

对比 swap 和 swapref 两个函数，并对比两个 main 函数，可以看到差别仅仅是：在 swapref 的函数头部的参数列表中，在形参前面加上 “&” 字符。

这样微小的书写差别产生了巨大的语义差别：调用函数时不再是使用传统的值传递机制了！在调用时不再是把实参的值复制给形参，而是把形参作为实参的别名，直接对实参进行操作。

需要提醒的是，对于使用引用作为形参的函数，在调用时产生了另一个限制，即对引用型的参数必须提供一个变量作为实参，而不能以常量或表达式作为实参。

例如下面的调用语句都是错误的：

```
swapref(10, 25); //wrong!
```

```
swapref(m, m + 12); //wrong!
```

思考一下程序运行结果：

```
#include <iostream>
using namespace std;
void myswap(int &a, int b) {
    int k=a;  a=b;  b=k;
    cout << "swap: a=" << a << ", b=" << b << endl;;
}
int main() {
    int m=10, n=25;
    cout << "before: m=" << m << ", n=" << n << endl;
    myswap(m, n);
    cout << "after: m=" << m << ", n=" << n << endl;
}
```

程序运行结果如何？

最后在此简单地提及常参数。

与常变量类似，函数也可以有常参数。这种参数同样由实参提供初值，但在函数体里不允许对它们重新赋值。常参数的定义形式也是在类型描述前加 `const` 关键字，如下所示：

```
int func(const int a, int b) { ... ... }
```

在这个函数的函数头部，参数 `a` 被指定为常参数，因此在函数体内不允许对它重新赋值。如果用户在函数体写了任何对 `a` 重新赋值的语句，则编译时就会出错。

第5章 函数与变量

5.1 函数的定义与调用

5.2 程序的函数分解

5.3 循环与递归

5.4 外部变量与静态局部变量

5.5 声明与定义

5.6 预处理

5.7 程序动态除错方法（二）

5.2 程序的函数分解

5.2.1 程序的函数分解



什么样的程序片段应该定义为函数：

1. 重复出现的相同/相似计算片段，可设法抽取共同性的东西，定义为函数。
 2. 长计算过程中有逻辑独立性的片段，即使出现一次也可定义为函数，以分解复杂性。
- 经验原则：可以定义为函数的东西，就应该定义为函数；一个函数一般不超过一页。
 - 往往存在很多可行的分解，寻找合理有效的分解是需要学习的东西。

5.2.2 函数封装和两种视角

函数封装

函数外部

关心的是函数如何使用：

- 函数实现什么功能
- 函数的名字是什么
- 函数有几个参数，类型是什么
- 函数返回什么值
-

函数头部的说明

函数内部

关心的是函数应当如何实现

- 采用什么计算方法
- 采用什么程序结构
- 怎样得到计算结果

....

封装把函数内外隔成两个世界。

不同世界形成了对函数的两种观点。

函数头规定了两个世界的交流方式。



- 函数是独立的逻辑实体。定义后可以调用执行。由此形成对函数的两种观察方式：

- 1) 从函数外（以函数使用者的角度）看函数；

- 2) 在函数内（以函数定义者的角度）看函数。



- 计划函数时，要同时从两个观点看：需要什么函数/参数/返回值？分析确定函数头部，定好公共规范。
- 写函数定义时应站在内部观点思考/解决问题；
- 使用函数时应站在外部立场上思考/解决问题。
- 功能描述清楚，接口定义好以后，函数定义和使用可由不同人做。要求双方遵循共同规范，对函数功能有一致理解。自己写函数时也要保证两种观点的一致性。

5.2.3 自定义函数示例

举例子让读者体会如何编写自定义函数和如何进行函数分解。希望通过这些例子说明在使用函数进行编程的一般技巧。读者也可以从中体会到函数分解的一般性经验。

【例5-10】以迭代公式求 x 的立方根

【例5-11】写函数求 $\sin x$ 的近似值

【例5-12】写函数判断变量 $year$ 的值是否闰年

【例5-13】写谓词函数判断质数

【例5-14】用函数验证歌德巴赫猜想

【例5-15】歌德巴赫(Goldbach)猜想+

【例5-10】求 x 立方根的迭代公式是 $x_{n+1} = \frac{1}{3}(2x_n + x/x_n^2)$ ，写一个程序，从键盘上输入 x 值，然后利用这个公式求 x 的立方根的近似值，要求达到精度 $|(x_{n+1} - x_n)/x_n| < 10^{-6}$ 。

解：把前文例题求出立方根的代码修改为自定义函数。

根据“cubic root”把函数命名为 `cbirt`，函数参数是一个 `double` 类型的数据，函数返回值即为求出的立方根。

```
double cbirt(double x) {  
    if (x==0)  
        return 0; //计算出0的立方根为0作为函数返回值  
    double x1, x2 = x;  
    do {  
        x1 = x2;  
        x2 = (2.0 * x1 + x / (x1 * x1)) / 3.0;  
        //cout << x2 << endl;  
    } while (fabs((x2 - x1) / x1) >= 1E-6);  
    return x2; //计算得到满足精度的项作为函数返回值  
}
```

写出 main 函数调用 cbrt 函数进行测试:

```
int main() { //测试cbrt
    double x;
    cout << "Input x to test cbrt(Ctrl-z to end)" << endl;
    while ((cin >> x))
        cout << "cbrt = " << cbrt(x) << endl;
    cout << "test finished." << endl;

    return 0;
}
```

运行时选择合理的测试数据: 0值/非0值, ± 1 , ± 8 , ± 27 ,
 ± 1000 , ± 1000000

【例5-11】 写一个函数利用公式 $\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$ 求出 $\sin x$ 的近似值（要求累加项的值小于 10^{-6} ），并与标准库中的 \sin 函数的计算结果进行比较。

把前文例题的源代码修改为自定义函数 dsin:

```
double dsin(double x) {  
    //x = fmod(x, 2*3.1415926); //此句有何作用?  
    double sum = 0.0, t = x;  
    int n = 0;  
    while (t >= 1E-7 || t <= -1E-7) {  
        sum = sum + t;  
        n = n + 1;  
        t = -t * x * x / (2*n) / (2*n + 1);  
        //cout << "n= " << n << " t= " << t << " sum=" << sum << endl;  
    }  
    return sum;  
}
```

```
int main() { //测试 dsin 函数。通过循环提供参数自动测试
    double x;
    cout << "test dsin:\nx\tdsin(x)\tsin(x)\n";
    for (int i=-100; i<=1000; i+=10) {
        //cout << "Please input x: ";
        //cin >> x;
        x = i;
        cout <<x << '\t' << dsin(x) <<'\t'<<sin(x) <<'\t';
        cout << dsin(x) - sin(x) <<endl;
    }
    return 0;
}
```

【例5-12】 写一个函数判断变量 year 的值是否表示一个闰年的年份，然后在main函数中调用这个函数，打印输出1900-2100中的闰年。

```
#include <iostream>
using namespace std;
```

```
int isleapyear(int year) {
    return ((year%4 == 0 && year%100 != 0) || year%400 == 0);
}
```

```
int main() {
    for (int year = 1900; year <= 2100; year++)
        if (isleapyear(year))
            cout << year << " ";
    return 0;
}
```

注意，这是两个不同的变量



【例5-13】写一个谓词函数，判断一个整数（参数）是否为质数。然后在 main 函数中判断并输出 -10 -- 999中的质数。

函数命名为 isprime，**注意把 $n \leq 1$ 时判断为非质数。**

把原有例题源程序改写为谓词函数 isprime:

```
int isprime(int n) { //版本1
    if (n <= 1) //n <= 1 时判断为非质数
        return 0;
    // n > 1 时继续分析判断
    int k;
    for (int k=2; k*k <= n; k++)
        if (n%k == 0) //发现一个因数就退出循环
            break;
    //根据循环退出或结束的情形来判断
    return (k*k <= n && n%k == 0)? 0:1;
}
```

函数可以更巧妙地使用 return 语句而写得更简洁：

```
int isprime(int n) { //版本2
    if (n <= 1 ) return 0;    //非质数
    for (int k = 2; k * k <= n; k++)
        if (n % k == 0) //发现一个因数就足以判断不是质数
            return 0;
    return 1; //上面循环中没有发现因数，所以判断是质数
}
```

有了上面的 isprime 函数，可以写出 main 函数如下：

```
int main() { //输出 -10--999之间的所有质数
    for (int n = -10; n < 999; n++)
        if (isprime(n))
            cout << n << " ";
    return 0;
}
```

可以把 isprime 函数和 main 函数拼装成一个完整的程序。

(要加上必要的其它内容；isprime 函数的两个版本只能任选其一)

注意在函数中对 $n \leq 1$ 进行了特殊处理： `if (n <= 1) return 0;`

由此可见，在写一个程序（或函数）**之前**，首先应该仔细分析需要考虑的情况。**完成之后**还应该仔细检查，看看是否有什么遗漏。如果事先分析周全，应该能看到这些问题。

从 isleapyear 和 isprime 这两个函数可以注意到，谓词函数通常只负责进行某种判断并返回判断结果，不进行信息输出。而由调用这类函数的程序根据自身需求进行信息输出。这是一种合理的函数功能分解方式。

```
int isleapyear(int year) {  
    return ((year%4 == 0 && year%100 != 0) || year%400 == 0);  
}
```

```
int isprime(int n) { //版本2  
    if (n <= 1 ) return 0; //非质数  
    for (int k = 2; k * k <= n; k++)  
        if (n % k == 0) //发现一个因数就足以判断不是质数  
            return 0;  
    return 1; //上面循环中没有发现因数，所以判断是质数  
}
```

【例5-14】回到例5-1，使用已有的 isprime 函数在小范围内验证歌德巴赫猜想：对 6 到 200 之间的各偶数找出一种质数分解，即找出两个质数，使它们的和等于这个偶数。
把 isprime 函数插入已写出的程序主体结构：

```
int main() {  
    int m, n;  
    for (m = 6; m <= 200; m += 2)  
        for (n = 3; n <= m/2; n += 2) {  
            //if ( n 是质数 && m-n是质数)  
            if (isprime(n) && isprime(m-n)){  
                cout << m << " = " << n << " + " << m-n << endl;  
                break;  
            }  
        }  
    return 0;  
}
```

请读者把 isprime 函数和这个 main 函数拼装成一个完整的程序文件。

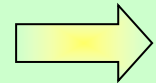
【例5-15】歌德巴赫(Goldbach)猜想“任一大于2的偶数都可写成两个质数之和”对于在计算机上已验证过的偶数都是成立的，而且很多偶数有多种分解方式，那么，对给定的偶数是否能找到一种分解方式，使分解得的两个质数之差小于该偶数的 $1/4$ ？

请写一个函数对给定的偶数寻找两个质数之差最小的分解方式，并把这两个质数返回到主调函数；

而且如果两个质数之差小于该偶数的 $1/4$ ，则函数的返回值不为0，表示成功；如果两个质数之差大于该偶数的 $1/4$ ，则函数的返回值为0，表示失败。

然后再写一个主函数，对6 ~ 200中的偶数进行验证是否可以这样分解。

- 寻找把偶数分解成两个质数之和，可以借鉴上一例题。
- 寻找“两个质数之差最小”的分解方式，只要从该偶数的 $1/2$ （准确地说是从“等于或大于该偶数的 $1/2$ 的第一个奇数”）开始往上搜索即可，所找到的第一种分解方式就满足“两个质数之差最小”。
- 题目中要求把给定的偶数进行分解成质数并返回“两个质数之差小于该偶数的 $1/4$ ”的判断结果，对此函数可以有多种设计方案。
- 先来看一种最直观的设计方案：



- (1)题目要求把给定的偶数分解成两个质数，可以把分解得的两个质数返回到主调函数，因此待编写的函数的形参设定为三个整数，用于表示待分解的偶数和分解而得的两个质数，而且分解而得的两个质数需要返回到主调函数中，所以这两个形参需要使用引用形式。
- (2)根据题目要求，显然要求函数的返回值是整数。
- (3)按照见名识义的原则，把函数命名为“goldbach”。

按照这一设计方案，写出函数头部如下：

```
int goldbach(int n, int &k1, int &k2)
```

用于分解的偶数

两个引用参数，表示进行
分解后得到的两个质数


```

int goldbach(int n, int &k1, int &k2) {
    if (n % 2 == 1 || n < 6)  //奇数或小于6的偶数不能分解
        return 0;
    k1 = (n / 2) % 2 ? n / 2 : n / 2 + 1; //等于或大于该偶数的1/2的第一个奇数
    for (k2 = n - k1; k1 <= n; k1 += 2, k2 = n - k1)
        if (isprime(k1) && isprime(k2))
            return (k1 - k2 < n / 4 ? 1 : 0);
}

```

用函数的返回值表示函数的工作状态

```

int main() {
    int m, m1, m2, found;
    for (m = 6; m <= 200; m += 2) {
        found = goldbach(m, m1, m2);
        cout << m << " = " << m1 << " + " << m2 << "\t";
        cout << (found? "Yes": "NO") << "\t" << m1 - m2 << endl;
    }
    return 0;
}

```

另一些可行的其它函数设计方案并编写程序——当然每一种函数设计方案都需要相应地考虑调用时如何处理：

(1) 把形参1也写成引用：

```
int goldbach1(int &n, int &k1, int &k2);
```

这种写法也可以。但在调用时必须对n提供一个变量作为实参，而不能直接写某个常数：

```
goldbach(1000, m1, m2); //ok
```

```
goldbach1(1000, m1, m2); //wrong!
```

(2) 只用 &k1就够了，不用 &k2：

```
int goldbach2(int n, int &k1);
```

这种写法也可以。但在主函数中调用后需要另行计算出第2个质数，如下所示：

```
found = goldbach2(m, m1);
```

```
cout << m << " = " << m1 << " + " << m - m1 << "\t";
```

(3) 不用引用，直接拿函数返回值表示一个质数（值为0时表示分解不成功）：

```
int goldbach3(int n);
```

这种写法也可以。但在调用时需要另想办法处理这两个质数，例如写成这样：

```
m1 = goldbach3(m);
```

```
cout << m << " = " << m1 << " + " << m - m1 << "\\t";
```

```
cout << (m1 ? "Yes": "NO") << endl;
```

由上例可知，同对一个问题，通常可以设计出多种函数设计方案来编程解答，每一种设计出的函数所需的参数、参数所代表的数据含义可以各不相同，相应地也需要在调用时加以考虑如何提供参数和使用函数返回值（或可以返回数据的参数）。

因此，读者应该理解，我们看到的各种成熟的函数，常常都是编程人员对多种可能的函数设计方案进行综合分析之后有所取舍而编写出来的。

我们在面对一个编程问题时，常常能构思出多种函数设计方案，这时需要学会取舍，选出其中一种自己觉得最为合理的方案来编程实现之。

对于上述例题的笔记

- 把原有的程序片段提取并改写成函数：把前面的代码中得到的数据改为函数的形参；把计算结果作为函数的返回值。
- 通常不在函数中向屏幕打印输出。以便让调用处进行后续计算或屏幕输出。
- 要根据调用时的需求来处理函数定义中的参数为值参数或引用参数。
- 每个问题都可能有多种函数分解方案，要选择最合适的方案进行处理。

上机练习内容:

1、例题 5-10, 5-11, 5-12, 5-13, 5-14, 5-15
(每一个写成单独的文件 xxxx-xxx-ex5-xx)

2、练习题5-2, 5-4。(每一个都写成单独的文件:
xxxx-xxx-prog5-x)

上交三个文件: 例题5-15, 练习题5-2和5-4

讲解了“5.2 程序的函数分解”之后，最好提前讲解“5.7 程序动态除错方法（二）”，然后再讲其它节。

第5章 函数与变量

5.1 函数的定义与调用

5.2 程序的函数分解

5.3 循环与递归

5.4 外部变量与静态局部变量

5.5 声明与定义

5.6 预处理

5.7 程序动态除错方法（二）

5.3 循环与递归

程序中有循环就可能导致很长的计算。

没有循环结构也能描述这类计算。C/C++ 语言允许递归，可在函数内调用自身，程序常常更简单清晰。

5.3.1 阶乘和乘幂（循环，递归）

【例5-16】 定义计算整数阶乘的函数：

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

乘的次数依赖于n，定义时未知，每次用可能不同。

类型特征可定为：

```
int fact(int)
```

阶乘值增长极快（数学），更合适的类型特征：

```
long fact(long)
```

可以用循环定义：

```
long fact1(int n) {  
    long i, f = 1;  
    for (i = 2; i <= n; ++i)  
        f *= i;  
    return f;  
}
```

程序的典型情况： 计算次数依赖于某些参数的值。

阶乘： $n! = 1 \times 2 \times \dots \times (n-1) \times n$

省略号不科学。严格定义需用递归形式。

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

递归定义的形式也提出了一种计算方法：

如果语言允许递归定义函数，就可直接翻译为程序。

C/C++ 允许递归定义：在函数定义内调用被定义函数本身。

阶乘函数的递归写法：

```
long fact (int n) {  
    return n == 0 ? 1 : n * fact(n-1);  
}
```

比循环函数简洁得多。

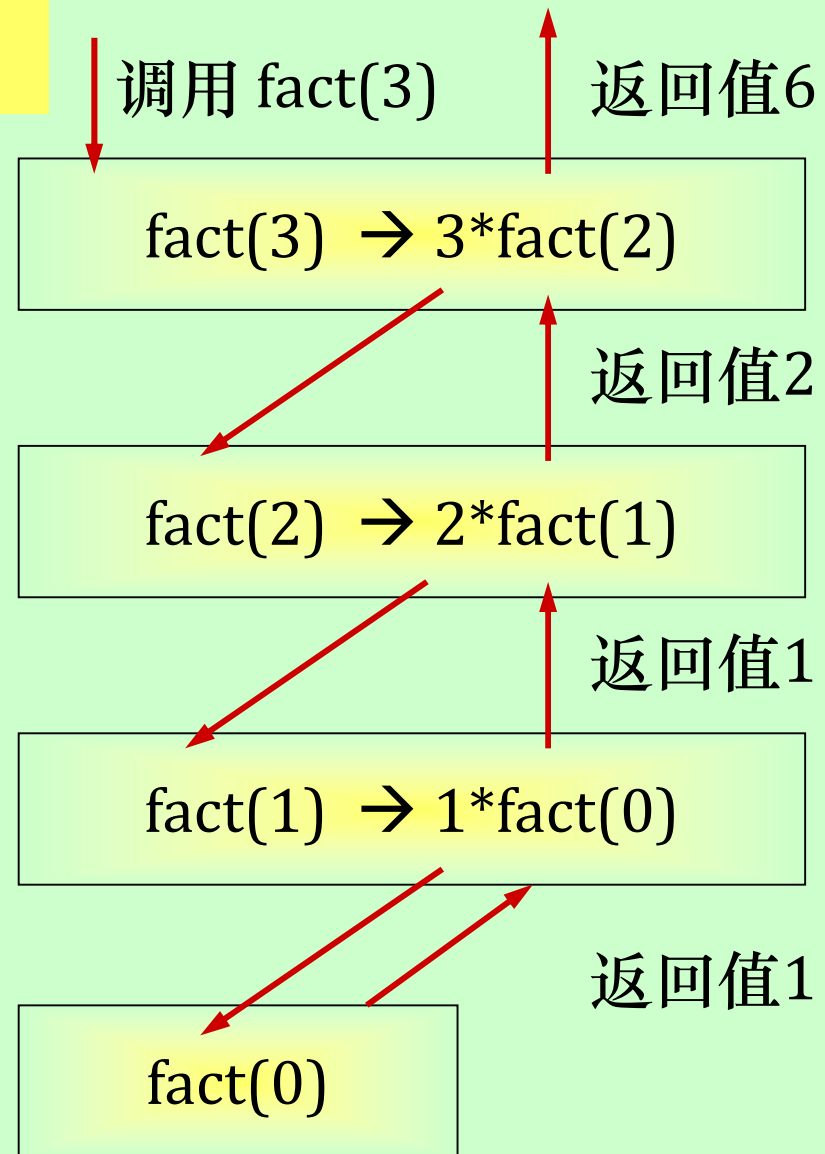
递归定义导致的计算过程

fact 实现的计算过程很不简单。

计算中 fact 被递归调用的次数由实参确定。参数不同，则递归调用次数（步数）不同。

考虑负参数值处理。可改为：

$n \leq 1 ? 1 : \dots$



fact(3) 的计算过程

【例5-17】递归求幂。写函数 `double dexp(int n)`

求 e （自然对数的底）的 n 次幂。

$$e^n = \begin{cases} 1 & n = 0 \\ e \times e^{n-1} & n > 0 \\ 1/e^{-n} & n < 0 \end{cases}$$

注意到参数 n 为负时乘幂也有定义。

先写一个辅助函数，再写一个所需的函数。

```
double dexp1 (int n) { //只处理 n>=0
    return n==0? 1: 2.71828*dexp1(n-1);
}
double dexp (int n) { // 分为 n>=0 和 n<0 处理
    return n>=0? dexp1(n) : 1/dexp1(-n);
}
```

这个问题也可以用循环写出（略）

```
long fact (int n) {  
    return n == 0 ? 1 : n * fact(n-1);  
}
```

```
double dexp1 (int n) {    //只处理 n>=0  
    return n==0? 1: 2.71828*dexp1(n-1);  
}
```

从上面例子可见，

递归的函数定义需要使用条件表达式或条件语句。

递归函数必须区分两种情况：

- (1) 直接给出结果的情况。是递归的基础
- (2) 需要递归处理的情况。其中把对较复杂情况的计算归结为对更简单情况的计算

循环与递归：有些循环程序也可以用递归的形式写；有些递归程序也可以通过循环写出。

当然，要写出一个递归函数，必须将其定义为函数（不能在main函数中写递归）。

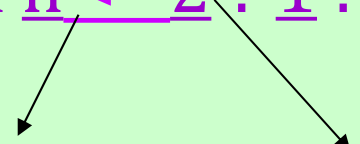
5.3.2 Fibonacci 序列（计算与时间）

【5-18】 Fibonacci（斐波那契）序列的递归定义：

$$F_1 = 1, F_2 = 1, \dots, F_n = F_{n-1} + F_{n-2} \ (n > 2)$$

递归函数定义：

```
long fib (int n) {  
    return n<=2 ? 1 : fib(n-1) + fib(n-2);  
}
```



负参数值也定义为 1。这是“合理”处置。

问题分析：这个程序好不好？

一方面，很好！程序与数学定义的关系很清晰，正确性容易确认，定义易读易理解。

写程序计时并分析：

```
int main () {  
    long t0, t1;  
    int n;  
    cout << "n \tfib(n) \ttime(s)" << endl;  
    for (n = 10; n <= 45; ++n) {  
        t0 = clock(); //调用 fib(n)之前的时刻  
        cout<< n <<"\t"<< fib(n) <<"\t"; //!!!  
        t1 = clock(); /////调用 fib(n)结束之后的时刻  
        cout << (double)(t1 - t0) / CLOCKS_PER_SEC  
            << endl;//时间差  
    }  
    return 0;  
}
```

n	fib(n)	time(s)
.....	(略)	
25	121393	0.001
26	196418	0.001
27	317811	0.002
28	514229	0.003
29	832040	0.004
30	1346269	0.007
31	2178309	0.01
32	3524578	0.016
33	5702887	0.025
34	9227465	0.04
35	14930352	0.064
36	24157817	0.103
37	39088169	0.168
38	63245986	0.268
39	102334155	0.437
40	165580141	0.701
41	267914296	1.136
42	433494437	1.834
43	701408733	2.969
44	1134903170	4.816

$$F_n = F_{n-1} + F_{n-2}$$

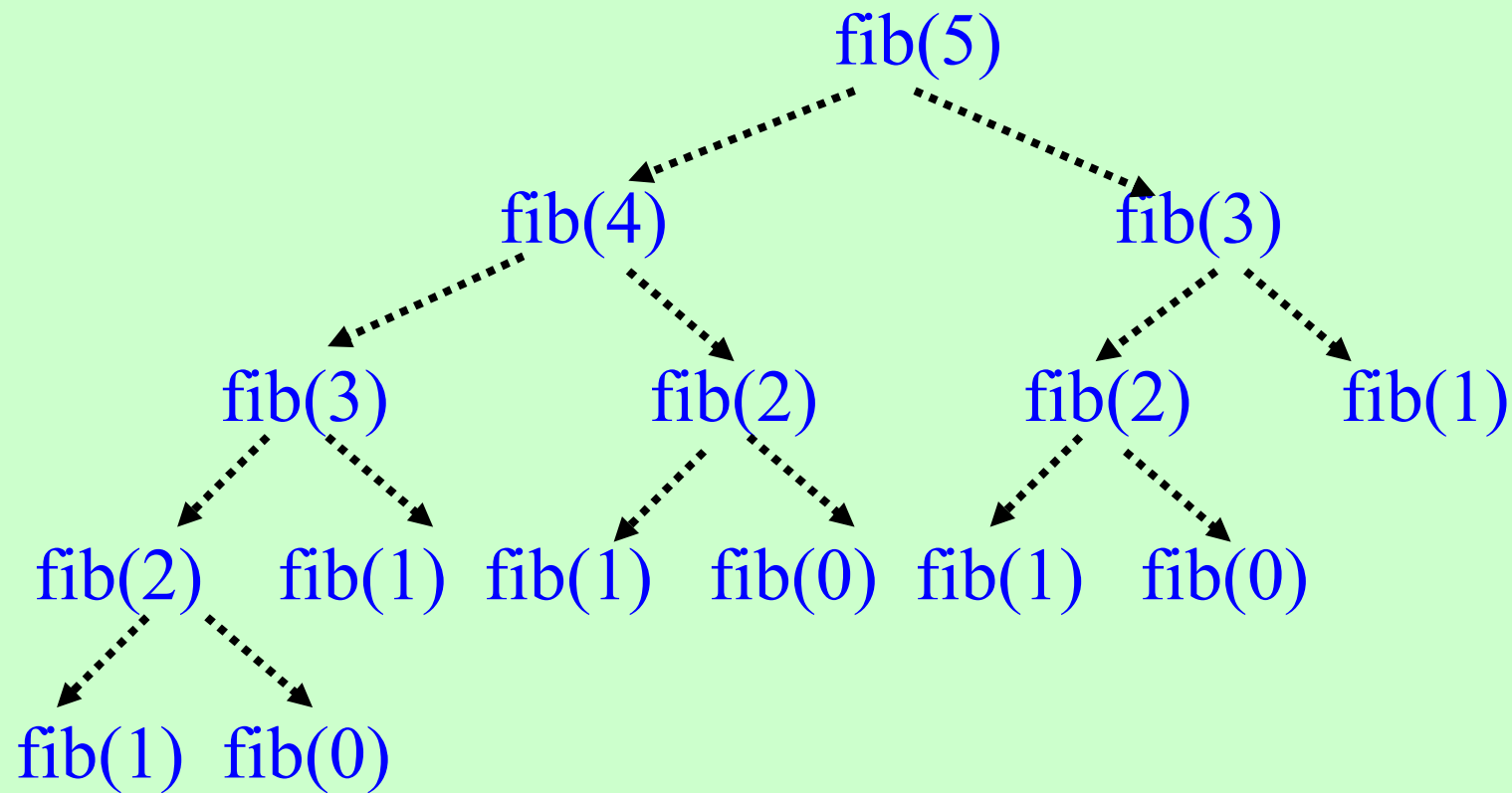
从数学定义上来看，n 加1时，Fn 只需要做一次加法就可以计算出来，

但函数 fib 计算所耗时间却明显增多（参数值增加1，函数 fib 的计算时间大约为原来的1.6倍，指数增长）！

为什么会这样呢？

存在着大量重复计算！ 参数越大重复计算越多。

示意图：



fib(5)计算中的函数调用情况

计算需时间，复杂计算需要很长时间。这是计算机的本质特征/弱点。说明它不万能，有些事情“不能”做。

人们发现了许多实际问题，理论上说可用计算机解决（可写出计算它的程序），但对规模大的情况（“大的参数 n ”），人根本等不到计算完成。

这时能说问题解决了吗？——不能。

理解这个情况对于理解计算机是非常重要的。

有一大类问题称为计算中的“难解问题”，其中有许多很实际的问题（规划、调度、优化等）。这方面的理论和实际技术的研究极为重要。

另外，对于许多问题的实用的有效算法，有极大的理论价值和实际价值。

二、用循环和尾递归求Fibonacci数列

```
long fibcycle (int n) { //循环方式计算Fibonacci数列的第n项
    long f1 = 1, f2 = 1, f3 = f1 + f2, k;
    if (n <= 2) return 1; //数列的前2项为1，当n < 0 时也返回1
    for (k = 4; k < n; ++k) { //n > 2时按通式递推计算
        f1 = f2;
        f2 = f3;
        f3 = f1 + f2;
    }
    return f3;
}
```

运算速度非常快

不能推论说递归函数就一定比用循环慢。

如果改用尾递归的方法编写求解Fibonacci数列的函数，其计算速度也可以很快。

```
long fib1 (long f1, long f2, int n) {  
    return n <= 1 ? f1 : fib1(f2, f1 + f2, n - 1);  
}
```

```
long fibtail (int n) {  
    return fib1(1, 1, n);  
}
```

5.3.3 求最大公约数（GCD）

【例5-19】用几种方法求整数的最大公约数（greatest common divisor）。

写函数 `long gcd(long m, long n)`

解法1：逐个检查，直到找到能同时整除 m 和 n 的最大整数（生成与检查）。需辅助变量 k 记录检查值。简单方式： k 顺序取值（初值/更新/结束），可用循环实现。

方式1： k 取初值 1 后递增，大于 m 或 n 时结束。

如何得到所需结果？ m 和 n 可能有多个公约数，最后的 k 值不是 m 和 n 的公约数（大于两数之一）。需要记录循环中找到的公约数。

只需记录已找到**最大的公约数**：用变量 **d**，初值1（是公约数），遇到新公约数（更大）时记入d：

```
if (m % k == 0 && n % k == 0)
    d = k; /* k为新找到的公约数 */
```

有了 d 及其初值，k 可以从2开始循环。函数定义：

```
long gcd (long m, long n) {
    long d = 1, k = 2;
    for (k=2 ; k <= m && k <= n; k++)
        if (m % k == 0 && n % k == 0)
            d = k;
    return d;
}
```

参数互素时初值 1 会留下来，也正确。

- 还有一些特殊情况需要处理：

- 1) m和n都为0需特殊处理。例如令函数返回值0；
- 2) 若m和n中一个为0，gcd是另一个数。函数的返回值正确。也可直接判断处理；
- 3) m、n为负时函数返回1，可能不对。

- 应在循环前加语句：

```
if (m == 0 && n == 0) return 0;
```

```
if (m < 0) m = -m;
```

```
if (n < 0) n = -n;
```

```
if (m == 0) return n;
```

```
if (n == 0) return m;
```

- (解法1) 方法2: 令 k 从某大数开始递减, 找到的第一个公约数就是最大公约数。k 初值可取 m 和 n 中小的一个。
- 结束条件: k 值达到 1 或找到了公约数。1 总是公约数。

```
for (k = (m > n ? n : m); m % k != 0 || n % k != 0; k--)  
    ; /* 空循环体 */  
return k; /*循环结束时k是最大公约数 */
```

- 本方法比前一方法简单一些。
- 两种方法的共同点是重复测试。这类方法的缺点是效率较低, 参数大时循环次数很多。

- 解法2：求GCD有著名的欧几里德算法（欧氏算法，辗转相除法）。最大公约数的递归定义：

$$\text{gcd}(m, n) = \begin{cases} n & m \bmod n = 0 \\ \text{gcd}(n, m \bmod n) & m \bmod n \neq 0 \end{cases}$$

函数定义（递归）：假设第二个参数非0，且参数都不小于0。与数学定义直接对应：

```
long gcd1 (long m, long n) {  
    return m%n == 0 ? n : gcd1(n, m%n);  
}
```

对欧氏算法的研究保证了本函数能结束，对较大的数计算速度也很快，远远优于顺序检查。

对特殊情况可另写一函数，其主体是对 gcd1 的调用：

```
long gcd(long m, long n) {  
    if (m < 0) m = -m;  
    if (n < 0) n = -n;  
    return n == 0 ? m : gcd1(m, n);  
}
```

- 函数定义2（循环方式）：辗转相除就是反复求余数，也是重复性工作，可用循环结构实现。
- 出发点 m 和 n ；循环判断 $m \% n$ 是否为 0，若是则 n 为结果；否则更新变量：令 m 取 n 的原值， n 取 $m \% n$ 的原值。为正确更新需用辅助变量 r ，正确的更新序列：

$r = m \% n; m = n; n = r;$

循环可写为：

```
for (r = m % n; r != 0; r = m % n) {  
    m = n; n = r;  
}
```

- 下面函数定义假定参数值不小于0（否则可以在前面增加判断和处理）：

```
long gcd2 (long m, long n) {  
    long r;  
    if (n == 0) return m;  
    for (r = m%n; r != 0; r = m%n) {  
        m = n; n = r;  
    }  
    return n;  
}
```

- 参数是局部变量，可在函数体里使用和修改。
- 请考虑，这里的循环不变式是什么？

关于GCD解法的小结：

- 解法1（生成与检查）

- 方法1： k 从1往上增加到m和n中的较小值
- 方法2： k 从m和n中的较小值往下减小到1

- 解法2（辗转相除法）

- 递归写法
- 循环写法

无论哪种解法和编程方法，都需要考虑为0或小于0的特殊情况。

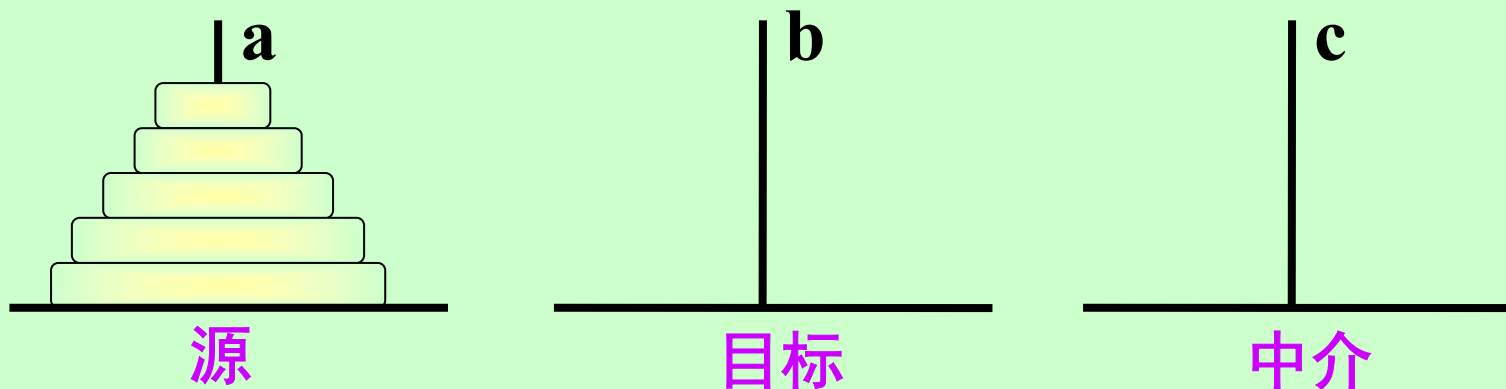
- 解法3（辗转相减法）： 练习题

5.3.4 梵塔问题

本问题用递归解决很简单，用循环解决较困难。

【例5-20】某神庙有三根细柱，64个大小不等、中心有孔的金盘套在柱上，构成梵塔。开始时圆盘从大到小套在一根柱上。目标是将所有圆盘从一柱移到另一柱。

规则：每次只移一个盘，大盘不能放到小盘上。
写程序模拟搬圆盘过程，打印出搬动指令序列。



初看问题似乎没规律。求解的关键在于看到问题的“递归性质”。搬 64 个盘的问题可归结为两次搬 63 个盘……。

一般情况：搬 n 个圆盘的问题可以归结为搬 $n-1$ 个圆盘。

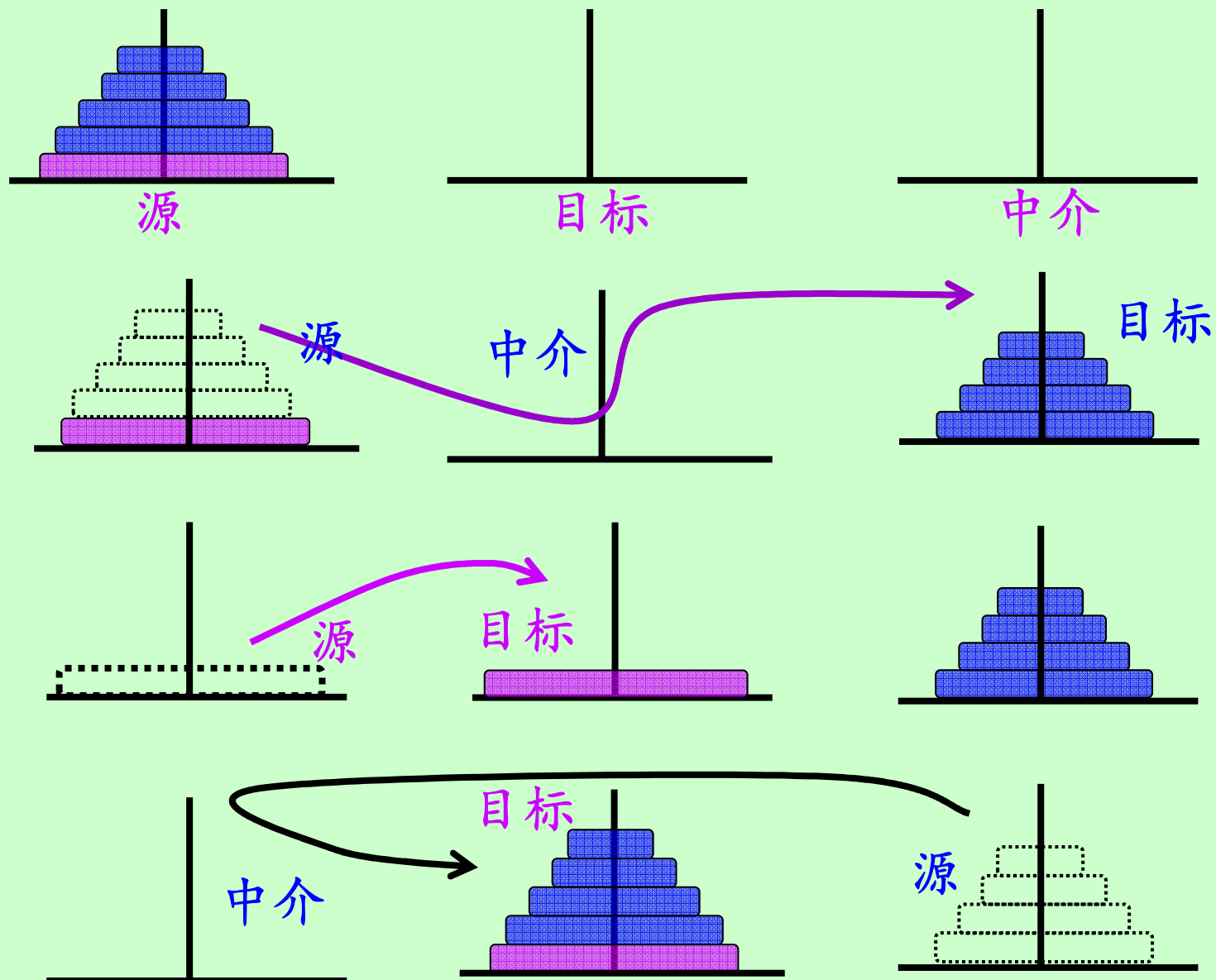
把 n 个盘从柱 a 搬到柱 b 的工作可以如下完成：

从柱 a 借助柱 b 将 $n-1$ 个圆盘搬到柱 c ；

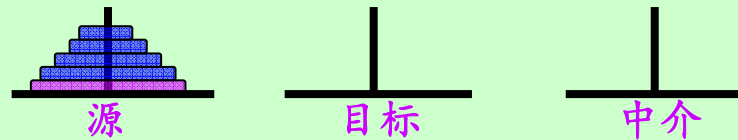
将最大圆盘从柱 a 搬到柱 b ；

从柱 c 借助柱 a 将 $n-1$ 个圆盘搬到柱 b ；

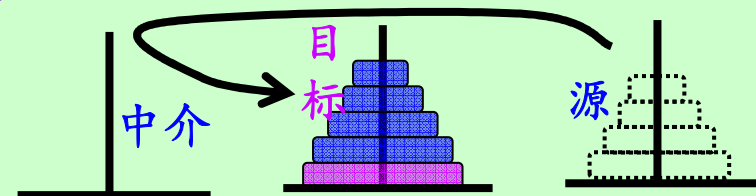
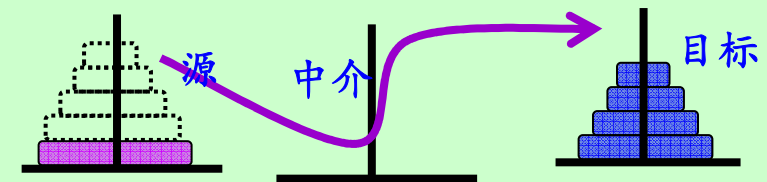
梵塔搬动示意图 hanoi(source, target, temp)



```
void moveone (int n, char source, char target) {
    //把第n号圆盘从源柱直接移动到目标柱。
    cout << n << ": " << source << " -> " << target << endl;
}
```



```
void hanoi(int n, char source, char target, char temp) {
    if (n == 1)
        moveone(n, source, target);
    else {
        hanoi(n-1, source, temp, target); //第1次递归调用 (搬n-1个)
        moveone(n, source, target);      //搬动第n号圆盘
        hanoi(n-1, temp, target, source); //第2次递归调用 (搬n-1个)
    }
}
```



函数调用: **hanoi(6, 'a', 'b', 'c');**

编写一个很简单的主函数来调用上面的函数：

```
int main() {  
    int n;  
    long t0, t1;  
    cout << "input n: ";  
    cin >> n;  
    t0 = clock();  
    hanoi(n, 'a', 'b', 'c');  //!!!  
    t1 = clock();  
    cout << "Finished. Time cost: "  
        << (double)(t1 - t0) / CLOCKS_PER_SEC;  
    return 0;  
}
```

有必要多次运行该程序，以理解执行过程。执行时先分别键入较小的 n 值（1、2、3、4 等等），观察屏幕上的输出信息；然后再键入较大的 n 值（10、20、30、60 等等），体会执行所需时间的变化。

课堂笔记： 5.3 循环与递归

5.3.1 递归函数：在定义函数时使用函数本身。

递归与循环可以互相转换。递归函数比较明显地显示出数学定义本身。

5.3.2 Fib序列 1, 1, 2, 3,... $F_n = F_{n-1} + F_{n-2}$

递归函数：return $n \leq 2 ? 1 : \text{fib}(n-1) + \text{fib}(n-2)$

优点：数学含义很清楚。缺点：存在大量重复计算。

改为循环实现，无重复计算，运行速度快。

循环与递归各有优缺点。

5.3.3 最大公约数

综合性的例题。

解法1 生成与检查。方法1（从小到大）比较直观。方法2（从大到小）使用了较多的C语言技巧。

解法2 辗转相除法（?）

$$\gcd(m, n) = \begin{cases} n & m \bmod n = 0 \\ \gcd(n, m \bmod n) & m \bmod n \neq 0 \end{cases}$$

函数定义1：递归写法。定义2：循环方式：（技巧高，比较难懂）

5.3.4 河内塔问题

用递归解决比较容易，但不容易用循环解决。

写递归函数时要注意每一次调用时的参数。

上机实验：

- 例题5-18 斐波那契序列的几种函数写法
- 例题5-19 求最大公约数的几种写法，并且在 main 函数中分别调用这几种写法。
- 练习题 5-6，5-7。
- 上交两个文件：XXXX-XXX-prog5-6-7（包含练习题 5-6和5-7，要在 main 函数中写出对两个函数的调用）。

请上传到QQ群文件的“6月30日第5章作业3”文件夹中。

截止时间：6月30日中午13:00

函数可看作对 **C/C++** 基本功能的扩充。

函数是特定计算过程的抽象，有通用性，可按规定方式（参数个数/类型）对具体数据使用。

例：标准函数 **sin**，类型特征是：

double sin(double)

标准函数有限，实际需求无限。

所以程序设计中有定义函数的需求。